Formalizing GDOI Group Key Management Requirements in NPATRL *

Catherine Meadows, Paul Syverson
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375-5320, USA
{meadows, syverson}@itd.nrl.navy.mil

Iliano Cervesato
Advanced Engineering and Sciences Division
ITT Industries, Inc.
Alexandria, VA 22303-1410, USA
iliano@itd.nrl.navy.mil

ABSTRACT

Although there is a substantial amount of work on formal requirements for two and three-party key distribution protocols, very little has been done on requirements for group protocols. However, since the latter have security requirements that can differ in important but subtle ways, we believe that a rigorous expression of these requirements can be useful in determining whether a given protocol can satisfy an application's needs. In this paper we make a first step in providing a formal understanding of security requirements for group key distribution by using the NPATRL language, a temporal requirement specification language for use with the NRL Protocol Analyzer. We specify the requirements for GDOI, a protocol being proposed as an IETF standard, which we are formally specifying and verifying in cooperation with the MSec working group.

1. INTRODUCTION

Before we can determine whether or not a security protocol satisfies its requirements, it is necessary to determine what those requirements are. Requirements are in general well understood for key distribution protocols involving two or three parties, and a number of formalizations of such requirements exist. But, they are not as well understood for group key distribution protocols, where keys may possibly be distributed among an arbitrarily large group of principals that may join or leave the group at any time. In this paper we attempt to fill this gap by developing a set of formal requirements for the GDOI group key management protocol [1], a protocol which we have been formally specifying and verifying as part of a joint effort with the IETF MSec working group. To do this, we used the NPATRL requirements language [16], a temporal language for cryptographic protocol requirements intended for use with the NRL Protocol Analyzer [10, 11]. What we found as a result of this effort was that requirements for group key distribution protocols were little understood, and that as much or more work needed to be put into developing a set of formal security requirements as into the formal specification of the protocol itself. Moreover, although these requirements were developed specifically for GDOI, we believe that they are generic enough so that they could be applied to many other group key management protocols with the appropriate modifications. The added complexity of the requirements resulting from the needs of secure group communication has also induced us to develop NPATRL into a full-scale logic that can be used to reason about requirements as well as specify them. We describe the logic and show how it can be applied in simplifying requirements.

A group key distribution protocol is one in which a key is distributed to members of a group, which may be of arbitrary size for some applications. This key may be distributed by the members of the group themselves (as in the Cliques protocol [13]), by a centralized key distributor, or by a collection of key distributors. Such protocols may support a number of different applications, such as secure multicast and secure dynamic coalitions. They usually must support the joining and leaving of members and possibly other operations. They may also put restrictions on joining members having access to old keys or on leaving members having access to new keys.

Superficially, group key distribution protocols seem to satisfy requirements very similar to pairwise key distribution protocols. Both have requirements for secrecy (no one outside the group should learn the key), authentication (recipients of the key should know where it came from and what it was intended for) and freshness (recipients should not be tricked into accepting an old key). But when we look at these requirements more closely, especially at secrecy and freshness, we see some important differences.

These differences arise from the fact that the notion of "session", which is so important to pairwise protocols, does not exist, at least not in the same sense, for group protocols. In a pairwise protocol, a session is determined by two communicating principals (possibly three, if a key server is used) and a key. Keys should not be learned by any other than the two or three communicating principals, and a key should be unique to a session. However, group protocols usually do not have such a notion of individual sessions strongly tied to principals and keys. Instead the paradigm is of a group which principals may enter and leave. Keys may be updated as principals enter or leave a group, in order that incoming principals have no access to old keys, or outgoing princi-

*Meadows and Syverson were supported by ONR. Cervesato was partially supported by NSF grant INT98-15731 "Logical Methods for Formal Verification of Software" and NRL under contract N00173-00-C-2086.

CCS'01, November 5-8, 2001, Philadelphia, Pennsylvania, USA.
ACM 1-58113-385-5/01/0011.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2001		**00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Formalizing GDOI Group Key Management Requirements in NPATRL	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**10**	

pals have no access to new keys. But keys may be updated for other reasons as well that have nothing to do with the composition of a group. Thus a freshness requirement that specifies a unique key per session will have no meaning here.

Likewise, the definition of 'secrecy' also needs to be reconsidered. In a pairwise protocol, all we require is that a session key only be known to the principals involved in that session. Whether or not one of the principals is dishonest and compromises the key is not usually a concern, as long as it is prevented from threatening the integrity of other sessions. However, in the case of group protocol, admitting a dishonest member into a group can introduce new risks. Not only can the member learn the group key while it is present, but depending upon how the protocol is designed, it may or may not be able to learn keys used before it joined the group, or keys generated after it left.

The rest of this paper is organized as follows. In Section 2 we give an overview of the GDOI protocol. In Section 3 we give an overview of the NPATRL logic, and describe the normal form for NPATRL requirements for the NRL Protocol Analyzer. In Section 4 we present the NPATRL requirements for GDOI, and we also describe our system for building secrecy requirements. Section 5 concludes the paper. In Appendix A we show how we were able to use the logic to remove recursiveness from the secrecy requirements and reduce them to normal form.

2. AN OVERVIEW OF GDOI

The GDOI (for Group Domain Of Interpretation) protocol [1] is intended to be used with the Internet Key Exchange (IKE) protocol [7, 5] to allow a Group Controller and Key Server (GCKS) to distribute keys to members of a group. Although it does not specify any mechanisms such as key hierarchies [2] for efficiently distributing keys to group members or for expelling or adding members, it is designed to be compatible with the use of such techniques. We have been working with the IETF MSec Working Group to develop a set of formal requirements, as well as a formal analysis, in order to demonstrate the usefulness of formal methods in the design or cryptographic protocols and in expediting the standardization process by providing formal evidence of soundness.

GDOI uses three categories of keys. *Category 1* keys are the pairwise keys shared between the GCKS and potential members. *Category 2* keys are key-encryption keys that are used to protect the *Category 3*, or traffic encryption keys.

For GDOI, the Category 1 (pairwise) keys are distributed via IKE Phase 1, which is described in [7, 5]. Key-encryption keys and traffic-encryption keys are created by the GCKS. The GCKS distributes these keys to the group as a whole by a *groupkey-push* message encrypted with the current key-encryption key. The GCKS maintains a sequence number `SEQ` that is incremented every time a new groupkey-push datagram is sent. The current value of the sequence number is included in the groupkey-push message. This allows group members to verify that a message is not a replay of one that they have already received. The groupkey-push datagram is also digitally signed by the GCKS using its private key so that receivers can verify that it was sent by the GCKS and not by another group member.

The groupkey-push message appears as follows in [1]:

Member		GCKS or Delegate
	⟵	`HDR*, SEQ, SA, KD, [CERT,] SIG`

The term `HDR*` indicates that everything is encrypted after the header, in this case using the current key encryption key. `SEQ` is the sequence number, `SA` the security association for this key payload, which gives such information as algorithms used, key lifetimes, etc., and `KD` the new keying material. `SIG` is the digital signature of the message, and `CERT` is an optional certificate for the signature key.

When a principal wants to join a group, it takes part in a four-message *groupkey-pull* exchange with the GCKS. All messages are encrypted and authenticated with the pairwise key shared between the two principals. In the first message, the principal sends a request to join the group, including the group identifier and a nonce `Ni` to help in verifying freshness.

The GCKS responds with its own nonce `Nr` and with the group Security Association, which describes the mechanisms (e.g. encryption algorithms) and policies used by the group. It holds off on sending the keying material itself until it can verify that the request is recent. The group member responds with a hash (`HASH(3)`) taken over the two nonces. The GCKS sends the keying material and the current value of the sequence number in the last message.

There are also some optional fields in the last two messages. If it is required by the group policy, the member can send its own part of a Diffie-Hellman key exchange in the third message (`KE_I`), and the GCKS can respond with its part of the exchange in the fourth message (`KE_R`). The resulting Diffie-Hellman key is used to encrypt the group keying material by use of exclusive-or. The purpose of this is to provide perfect forward secrecy: even if a pairwise key is compromised, the intruder can learn only keys distributed after the compromise, not those distributed before.

Another option allows the two principals to verify that each is authorized to act in their respective roles. This is the proof-of-possession (POP) option, where each party includes a public key certificate signed by a relevant authority, and proves his or her possession of the key by using it to sign the two nonces that were exchanged earlier in the protocol.

The four messages sent in the groupkey-pull exchange appear as follows in [1]:

Initiator (Member)		Responder (GCKS)
`HDR*, HASH(1), Ni, ID`	⟶	
	⟵	`HDR*, HASH(2), Nr, SA`
`HDR*, HASH(3) [, KE_I]` `[,CERT] [,POP_I]`	⟶	
	⟵	`HDR*, HASH(4), [KE_R,] SEQ,` `KD [,CERT] [,POP_R]`

where `Ni` and `Nr` are the two nonces, `SA` is the security association, `KE_I` and `KE_R` are the optional Diffie-Hellman halves, `CERT`, `POP_I`, `POP_R` are are the certificates and signatures used in the optional proof-of-possession exchange, and `SEQ` and `KD` are the sequence number and keying material (encrypted with the Diffie-Hellman key if that is used), respectively. The notation `HDR*` means, as before, that all information after the header is encrypted, this time with the shared Category 1 key. The hashes in the exchange are computed over the information sent in the respective messages. More detail may be found in [1].

Note that in no place does GDOI specify means for eliminating members from the group. This is accomplished using

something called a *key hierarchy*. Basically, a key hierarchy is a tree, the root of which is the actual key used for encryption. Nodes of the tree encrypt and authenticate the nodes above it. When a principal is admitted to the group, it is assigned a leaf of the tree. When it leaves the group, only the (limited) portion of the tree it needs to compute the group key ough to be updated. This allows access control for both entering and leaving members to be enforced in an efficient way, as well as providing extra security beyond that provided by the key-encryption key used to encrypt the push message, since a new key will be protected by the keys below it in the hierarchy. See [2] for a discussion and overview of key hierarchies.

3. THE NPATRL LOGIC

3.1 The NRL Protocol Analyzer Model

The NRL Protocol Analyzer, or NPA for short, is a computer-assisted verification tool for security protocols which combines model checking and theorem-proving techniques to establish authentication and secrecy properties. We present merely a brief overview here. The interested reader is invited to consult [10, 11] for further details.

A protocol is modeled as a number of communicating state machines, each associated with a different roles. Their transitions correspond to the actions that comprise the corresponding role. At run time, roles are executed by *honest principals* who faithfully follow the protocol. Several instances can be executing at the same time, and they are distinguished by means of a unique round number. The intruder is modeled after the Dolev-Yao adversary [4]. *Dishonest principals* share their keys and other confidential information with the adversary.

The messages in transit, the information held by each principal and the intruder, the runs currently being executed, and the point that each of them has reached constitute the global *state* for the NPA. A protocol action implements a local transformation with global effects on the state. The initial state is implicit in the protocol specification.

In order to verify a protocol, a specification is fed into the run-time system of the NRL Protocol Analyzer together with the description of a family of states that correspond to attack situations. The system applies protocol actions backwards from these target states until it either reaches the initial state, or it exhausts all possibilities for doing so. As it regresses back towards the initial state, the NPA maintains a *trace* of the sequence of actions that, when executed, lead to the target state. If the initial state is ever reached, the resulting trace is a potential attack. If all possibilities are exhausted, there is no attack of the kind sought. Although the search space is in general infinite, the NPA incorporates techniques based on theorem proving that have the effect of soundly restricting the search to a finite abstraction, in most cases.

Traces are sequences of *events* of the following form:

$$event(P, Q, T, L, N)$$

In general, any protocol or intruder state transition may be assigned an event. The arguments are interpreted as follows: P is the principal executing the transition, Q is the set of the other parties involved in it, T is a name that identifies the transition, L is a set of relevant words, and N is the local round number of the transition. Typical categories of events correspond to receiving a message, accepting data as valid as a result of performing certain checks and sending a message. For example:

$$event(\mathsf{user}(A, \mathsf{honest}), [\mathsf{user}(B, H)], \mathsf{initiator\_accept\_key}, [K], N)$$

This event describes the execution of a transition called "initiator_accept_key" by honest principal A that involves a key K and some other principal B who may or may not be honest.

3.2 The NPATRL Syntax

The NRL Protocol Analyzer has successfully analyzed a number of protocols, sometimes uncovering previously unknown flaws [10, 11]. But, secrecy and authentication goals are awkwardly expressed, as states that should not be reachable from the initial state. This unintuitive and occasionally error prone way of writing requirements would have made it very difficult to use the NPA for large protocols.

The *NRL Protocol Analyzer Temporal Requirements Language*, better known as NPATRL (and pronounced "N Patrol"), was designed to address these shortcomings [16]. This formalism makes available the abstract expressiveness of a logical language to specify requirements at a high enough level to capture intuitive goals precisely, and yet it can be interpreted in the NPA search engine.

NPATRL requirements are logical expressions whose atomic formulas are *event statements*, which mostly correspond to events in the NRL Protocol Analyzer; they include events denoting actions by honest principals that can be found in the trace of an NPA search, and the special learn event that indicates the acquisition of information by the adversary. NPATRL's syntax for events is similar but not identical to the NPA's. In NPATRL, the NPA accept event given above is written:

$$\mathsf{initiator\_accept\_key}(\mathsf{user}(A, \mathsf{honest}), \mathsf{user}(B, H), K, N)$$

The logical infrastructure of NPATRL consists of the usual connectives $\neg$, $\wedge$, $\rightarrow$, etc, and the temporal modality $\Diamond\!\!\!-$ which is interpreted as "happened at some time before" or "previously".

For example, we may have the following requirement:

> *If an honest principal A accepts a key K for communicating with another honest principal B, then a server must have previously generated and sent this key with the idea that it should be used for communications between A and B, and that both are expected to be honest.*

We can use NRL Protocol Analyzer events to construct an NPATRL formula that expresses it:

$$\begin{aligned} &\mathsf{initiator\_accept\_key}(\mathsf{user}(A, \mathsf{honest}), \mathsf{user}(B, H), K, N) \\ \rightarrow\ &\Diamond\!\!\!- \ \mathsf{svr\_send\_key}(\mathsf{server}, (\mathsf{user}(A, \mathsf{honest}), \mathsf{user}(B, \mathsf{honest})), K, N) \end{aligned}$$

This formula is a simple expression of the above requirement.

Intuitively, the protocol verification process changes from what we discussed in the previous section by using NPATRL requirements where the final state appeared. More precisely, we first need to map every NPATRL event statement to an actual event in the NPA specification of the protocol. Then, we take the negation of each NPATRL requirement as a way to characterize the states that should be unreachable if and only if that requirement is satisfied. At this point, we perform the analysis as in the previous section: if the NPA proves that this goal is unreachable, the protocol satisfies the original requirement. Otherwise, it returns a trace

corresponding to an attack on the protocol that potentially invalidates the requirement.

A couple of particular points about NPATRL expressions: Events occur exactly once. This means that atomic formulas are true at exactly one point in a trace (if at all). There is nothing in NPATRL syntax to automatically guarantee this uniqueness; it is assumed that event statements contain enough individuating information in their arguments or predicate to enforce this. Note that NPA guarantees this uniqueness, in part by having all events indexed both by local runs and timestamps. Second, "$\diamondminus$" is a strict operator; it includes times prior to the present time but does not include the present time. It is also convenient, especially when stating axioms, to have the dual operator in our language, "$\boxminus$", read as "at all previous times" or "always previously". It can be defined logically by, $\boxminus\varphi \leftrightarrow \neg\diamondminus\neg\varphi$, where φ is an formula.

NPATRL has been extensively used in the last few years to analyze protocols with various characteristics. Among these, generic requirements have been given for two-party key distribution protocols [14, 15] and two-party key agreement protocols [16]. The most ambitious specification undertaken using NPATRL has involved the requirements of the credit card payment transaction protocol SET (Secure Electronic Transactions) [9]. SET proved particularly difficult to specify for several reasons. One of these was that the objects to be authenticated are dynamic: unlike keys, what is agreed upon changes as it passes from one principal to another. This exercise revealed several ambiguities [9].

Our current task, formalizing group key management requirements, has its own dynamics. Even when the data objects (keys) are constant, the principals sharing them are not. And the very notion of a session is much less well defined than in previously studied cases. Perhaps most significantly, until this point we had been able to use NPATRL as just a language. All statements were interpreted into the NPA and evaluated there. However, we have found it necessary to reason at the level of NPATRL itself. This requires a logic for our logical language.

3.3 NPATRL Axioms

We give axioms of a normal modal logic adequate to capture the needed temporal reasoning. Readers are referred to standard texts for details on systems of modal and temporal logic [3, 6, 8].

Our logic has two inference rules:

Modus Ponens: From φ and $\varphi \rightarrow \psi$ infer ψ.

Necessitation: From $\vdash \varphi$ infer $\vdash \boxminus\varphi$.

'$\vdash$' is a metalingusitic symbol. '$\Gamma \vdash \varphi$' means that φ is derivable from the set of formulae Γ (and the axioms as stated below). '$\vdash \varphi$' means that φ is a theorem, i.e., derivable from axioms alone. Axioms are all instances of tautologies of classical propositional logic, and all instances of the following axiom schemata

K $\boxminus(\varphi \rightarrow \psi) \rightarrow (\boxminus\varphi \rightarrow \boxminus\psi)$

4 $\boxminus\varphi \rightarrow \boxminus\boxminus\varphi$

W $\boxminus(\boxminus\varphi \rightarrow \varphi) \rightarrow \boxminus\varphi$

L $\boxminus((\varphi \wedge \boxminus\varphi) \rightarrow \psi) \vee \boxminus((\psi \wedge \boxminus\psi) \rightarrow \varphi)$

The first axiom guarantees that our temporal operators respect the non-temporal part of the logic. The second one guarantees that temporal reasoning is transitive. The third guarantees that sets of events are always finite and strictly-ordered. The last guarantees that events are weakly-connected (comparable). Note that in the presence of **K** and **W**, the **4** axiom becomes redundant. We have explicitly included it because we specifically use transitivity in our arguments in appendix A. There is some discussion of logics containing these axioms in [6] and [8]. In [8], axiom **L** is called "$\mathbf{Lem}_0$" after Lemmon. Space limitations preclude a more detailed presentation here.

3.4 NPA Acceptable Expressions

Although NPATRL was originally designed to be used with the NRL Protocol Analyzer, it is actually much more expressive than the set of specifications accepted by the tool. Thus, in order to make NPATRL usable with NPA, it is necessary to identify a subset of NPATRL requirements that are acceptable by NPA, and to put them into a normal form that is parsable by NPA.

An NRL Protocol Analyzer query can be specified in terms of three things: terms known by the intruder, values of local state variables, and sequences of events that did or did not occur. The part of the query concerning sequences of events corresponds most closely to the NPATRL events. However, these events only correspond to user actions, and do not include learn events, which correspond to intruder actions. In order to capture intruder learn events, we will need to make use of the part of the query that specifies terms known by the intruder. This can cause some difficulties, since an NPA query does not specify when the terms were learned by the intruder. However, we can simplify matters by limiting ourselves to queries which specify the learning of only one term. This is usually adequate, and when it is not, we can usually transform the NPATRL requirement using our logic so that the restriction is satisfied.

Given that, we specify a normal form **R** for NPA acceptable expressions in a BNF grammar as follows. We let w stand for any learn event, a stand for any atomic event that is not a learn event, and let b stand for any atomic event.

$$
\begin{array}{lcl}
\mathbf{E} & ::= & \diamondminus a \mid \diamondminus(a \wedge \mathbf{E}) \\
\mathbf{F} & ::= & \mathbf{E} \mid \mathbf{E} \vee \mathbf{F} \mid \mathbf{E} \wedge \mathbf{F} \\
\mathbf{G} & ::= & \neg\mathbf{E} \mid \neg\mathbf{E} \wedge \mathbf{G} \\
\mathbf{R} & ::= & \neg b \mid a \rightarrow \mathbf{G} \mid b \rightarrow \mathbf{F} \mid a \rightarrow \mathbf{G} \vee \mathbf{F} \mid a \rightarrow \mathbf{G} \wedge \mathbf{F} \\
 & & \mid \diamondminus w \rightarrow \mathbf{G} \mid \diamondminus w \rightarrow \mathbf{G} \vee \mathbf{F} \mid \diamondminus w \rightarrow \mathbf{G} \wedge \mathbf{F}
\end{array}
$$

4. REQUIREMENTS FOR GDOI

4.1 Assumptions

We assume that each group is managed by one GCKS (it is possible to have more, but the means for doing this are not specified in the GDOI document). We assume that a GCKS may manage more than one group, and that a member may belong to more than one group. We assume that members may both join and leave a group, and a member may have concurrent and/or overlapping memberships in the same group.

We assume the usual Dolev-Yao style intruder, who can read, alter, destroy, and create traffic, and is in league with any dishonest principals, who share all data with it. We assume that all GCKSs are honest, but that some members may be dishonest. Note that as a result of this assumption we make no distinction between the intruder's learning a

key and a principal learning a key to which it is not entitled. Only dishonest principals will attempt to gain access to keys to which they are not entitled, and dishonest principals are assumed to share all information with the intruder.

We assume that there are two ways in which a key can be compromised that cannot be prevented by the protocol. One is by stealing: the intruder may learn the key by cryptanalysis, theft, etc. even if all possessors of the key are honest. The other is by having a dishonest member join the group.

Finally, in order to simplify matters, we only define events and requirements for key encryption keys, not traffic encryption keys. Since traffic encryption keys are protected by key encryption keys and distributed via the same mechanisms, it should be relatively straightforward to derive their requirements from the requirements for key encryption keys.

4.2 GDOI Events

In general, events map to actual messages and vice versa. However, since the central messages of the groupkey-pull exchange simply defer computation in order to resist forms of denial-of-service attacks [12] and the NPA does not currently support reasoning about denial-of-service, we behave as if their information load were compounded with the outer messages of this exchange.

We divide the possible GDOI events according to the principals that engage in them. There are four types of principals: the intruder, the GCKS, the group member, and an authorization server responsible for issuing credentials. Since a group member may be honest or dishonest, we represent a general group member as $\mathsf{member}(M, H)$, an honest member as $\mathsf{member}(M, \mathsf{honest})$, and a dishonest group member as $\mathsf{member}(M, \mathsf{dishonest})$.

4.2.1 Intruder Events

There is only one intruder event of interest to us here: the event in which the intruder P learns a word W. We represent that as follows:

$$\mathsf{learn}(P, (), W, N)$$

4.2.2 Authorization Server Event

The authorization server is responsible only for issuing credentials to principals. In order to simplify matters, we assume that each group has its own set of credentials appropriate to it.

This action is represented by

$$\mathsf{auth\_issuecreds}(AUTH, X, (C, G), N)$$

where X is the principal to whom the credentials are issued, C stands for the credentials, and G is the group to which the credentials apply.

4.2.3 The GCKS

The GCKS performs a number of actions of interest. It can create a key encryption key. It can admit and expel members. It can also cause a key to become current, and cause a key to expire. It can send a key, either in response to a member's request, or as part of a group-key push datagram. We represent these as follows:

Creating a key:

$$\mathsf{gcks\_createkey}(GCKS, (), (G, K_G), N)$$

where G is the group for which $GCKS$ is creating the key encryption key, K_G.

Sending a key as a result of a pull exchange:

$$\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, N_M, N_{GM}, G, K_{GM}), N)$$

where M is the member, K_G is the key, G is the group, K_{GM} is the pairwise key, N_M is the nonce M uses in initiating the exchange and N_{GM} is the nonce G uses in responding. We also use the gcks_sendpullkey event to cover the GCKS's admitting M to the group, since M requests membership by initiating a pull protocol. We use N_{GM} to identify M's particular membership in the group. Note that this may not be the identifier used in a real application (as a matter of fact, GDOI does not specify any kind of membership identifier); however it is useful from a requirements point of view in that it allows us to distinguish between different and possibly overlapping memberships on the part of the same individual.

Sending a key in a push message:

$$\mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), N)$$

The event gcks_sendpushkey causes one key, K'_G, to expire for group G and causes the next one K_G to become current. The initial key created for a group is first sent in a pull-key message. Except for such initial keys, we assume for convenience that a push-key message making K_G current is sent immediately after the create event that produced K_G (without the possibility of an intervening distribution in a pull-key message). We also assume that the initial key is sent in at least one pull-key response that takes place immediately after its creation. We say the initial key becomes current when that first pull-key response containing it is sent.

We note that neither gcks_sendpullkey nor gcks_sendpushkey tell the whole story about the keying material passed in these two messages. In actual fact, the pull-key message will contain, not only the current key, but also the relevant part of the key hierarchy that the member M needs to access the key. Likewise the gcks_sendpushkey message will also contain the portion of the key hierarchy that needs to be changed to give members access to the new key and prevent former members from accessing the new key, if this is desired.

Canceling a membership:

$$\mathsf{gcks\_cancel}(GCKS, M, (G, N_{GM}), N)$$

where M is the member, G is the group, and N_{GM} identifies the membership. Note that expulsion cancels only the membership with identifier N_{GM}, not all memberships of that member. In order to truly expel the member, all its memberships would have to be canceled.

We note that gcks_cancel would be achieved in GDOI by having the GCKS send out a push message containing a new key hierarchy from which M is excluded. We choose to specify gcks_cancel separately from gcks_sendpushkey since this allows us to avoid issues such as canceling multiple memberships in one message, etc.

Sending a POP:

$$\mathsf{gcks\_sendpop}(GCKS, M, (G, N_{GM}, N_M, C_G), N)$$

This event describes a GCKS sending a POP in response to a members request. C_G stands for G's credentials.

Stealing a Key:

Finally, we need to specify the stealing of a key. We think of this not as something done by the intruder, but as something done by the GCKS. In other words, the action of stealing a key needs to be precipitated by the GCKS "losing" a key. This appears paradoxical, but it is a result of our model's assumption that actions involving a piece of data can only be initiated by those in possession of it. Note that we could also include actions describing members losing keys, but that this would be redundant.

We have two event statements, one describing the loss of a key-encryption key, and one describing the loss of a pairwise key:

$$\mathsf{gcks\_losegroupkey}(GCKS, (), (G, K_G), N)$$

where G is the group and K_G is the key.

$$\mathsf{gcks\_losepairwisekey}(GCKS, (), (GCKS, M, K_{GM}), N)$$

where K_{GM} is the pairwise key and M is the member who shares the key with the GCKS.

4.2.4 Member Actions

The relevant member actions involve accepting a key and requesting a key. A member can only request a key by initiating a group-key pull exchange, but it may accept a key as a result of receiving the final message of a group-key pull exchange or as a result of receiving a group-key push message.

The events are specified as follows.

Requesting a Key:

$$\mathsf{member\_requestkey}(M, GCKS, (G, N_M, K_{GM}), N)$$

where $GCKS$ is the GCKS, G is the group, N_M is the nonce M uses in initiating the request (to distinguish it from other requests), and K_{GM} is the pairwise key shared between $GCKS$ and M. This corresponds to M sending the first message in a group-key pull exchange.

Accepting a Key From a Group-key Pull Exchange:

$$\mathsf{member\_acceptpullkey}(M, GCKS, (G, K_G, N_{GM}, N_M, K_{GM}), N)$$

where $GCKS$ is the GCKS, G is the group, K_G is the key, and K_{GM} is the pairwise key shared between M and the GCKS. Again, this does not give the whole picture, as it leaves out the portion of the key hierarchy that the GCKS sends to M.

Accepting a Key From a Group-key Push Message:

$$\mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, K'_G), N)$$

where $GCKS$ is the GCKS, G is the group, K_G is the new key, and K'_G is the current key encryption key. Again, we leave out the portion of the key hierarchy that M uses to authenticate the message and decrypt K_G.

For conciseness, we set

$$\begin{aligned}&\mathsf{member\_acceptkey}(M, GCKS, (G, K_G), N)\\ \leftrightarrow\ &\mathsf{member\_acceptpullkey}(M, GCKS, (G, K_G, \_, \_, \_), N)\\ &\vee \mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, \_), N)\end{aligned}$$

Here and in the rest of this paper, we write "_" for an argument whose actual value is irrelevant. Each occurrence can be thought of as a distinct variable.

Sending a POP:

$$\mathsf{member\_sendpop}(M, GCKS, (G, N_M, N_{GM}, C_M), N)$$

This event describes a member M sending a POP in response to a GCKS's request. C_M stands for M's credentials.

4.3 Authentication Requirements

Since GDOI is only intended to address the problem of secure distribution of group keys, not the authentication of group members to each other, its authentication requirements are simple and rather similar to those for two-party protocols. Thus we give them first. There are authentication requirements for both the group member and the GCKS. The group member will want to know, if it accepts a key, that that key was generated by the GCKS for that group. The GCKS will want to know that, if it sends a key to a group member, then that group member requested a key. Finally, there are authentication requirements on the proof of possession (POP) algorithm. If the GCKS accepts a proof of possession from a group member, then the group member should have obtained the appropriate authorizations and the group member should have responded to the GCKS's request for a POP. A similar requirement holds for the group member's accepting a POP from the GCKS. We consider the three types of authentication below.

4.3.1 Authentication of a Key to a Group Member

Since there are two different ways a group member can receive a key, we have two different sets of requirement. In the case of the group member M accepting a key K_G for group G as a result of the pull protocol, we require that one of two things must have happened; either the pairwise key shared between the member and the GCKS was lost, or the GCKS did send K_G to M for use in G:

$$\begin{aligned}&\mathsf{member\_acceptpullkey}(M, GCKS, (G, K_G, N_M, N_{GM}, K_{GM}), \_)\\ \rightarrow\ &\diamondminus\mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)\\ &\vee \diamondminus\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, \_, \_, G, K_{GM}), \_)\end{aligned}$$

In the case of the member M accepting the key K_G for group G as the result of receiving a push datagram, we again require that the GCKS has sent K_G for use in G in a push datagram protected by key K'_G:

$$\begin{aligned}&\mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, K'_G), \_)\\ &\rightarrow \diamondminus\mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), \_)\end{aligned}$$

Observe that this requirement does not make any provision for losing an old group key K'_G since gcks_sendpushkey messages are authenticated with the signature of the GCKS.

4.3.2 Authentication of a Group Member's Request

Although the GCKS has two ways of sending keys, it has only one way of sending a key to a specific group member: via a pull protocol. Thus we need only one requirement here, saying that if the GCKS sent a key to a group member in response to a pull protocol request, then either the pairwise key between the GCKS and group member was lost, or the group member actually sent that request. We need a unique way of identifying the group member's request, and so we will use the nonce the group member sends in the first message of the pull protocol:

$$\begin{aligned}&\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, N_M, N_{GM}, G, K_{GM}), \_)\\ \rightarrow\ &\diamondminus\mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)\\ &\vee \diamondminus\mathsf{member\_requestkey}(M, GCKS, (G, N_M, K_{GM}), \_)\end{aligned}$$

4.3.3 Authentication of a Proof of Possession

For proofs of possession, we want to show that, for either the GCKS or a member, if A accepts a key requiring a proof of possession from B, then B sent the POP in response to

A's request, and B obtained the credentials from the appropriate authority. The act of obtaining credentials is outside the scope of GDOI; however, we leave it in the requirement specification because it is clearly the intent of the POP.

$$\begin{array}{l}\mathsf{gcks\_sendpullkey}(GCKS, M, (K_G, N_M, N_{GM}, G, K_{GM}), \_)\\ \rightarrow \Diamond\!\!-\ (\quad \mathsf{member\_sendpop}(M, GCKS, (G, N_M, N_{GM}, M, C_M), \_)\\ \qquad \wedge\ \Diamond\!\!-\mathsf{auth\_issuecreds}(AUTH, M, (C_M, G), \_))\end{array}$$

$$\begin{array}{l}\mathsf{member\_acceptpullkey}(M, GCKS, (K_G, N_M, N_{GM}, G, K_{GM}), \_)\\ \rightarrow \Diamond\!\!-\ (\quad \mathsf{gcks\_sendpop}(GCKS, M(G, N_M, N_{GM}, C_M), \_)\\ \qquad \wedge\ \Diamond\!\!-\mathsf{auth\_issuecreds}(AUTH, M, (C_M, G), \_))\end{array}$$

4.4 Freshness Requirements

For GDOI, we can identify two types of freshness. One, we call *recency freshness*. This is the requirement that, if a principal receives a piece of information, such as a key, then it must have been current at some specified point in time according to the principal's local clock, for example when the principal requested it. The other, we call *sequential freshness*. This is the requirement that, if a principal accepts a key K_G, then it could not have previously accepted a key that became current after K_G.

4.4.1 Recency Freshness for Pull Protocol

$$\begin{array}{l}\mathsf{member\_acceptpullkey}(M, GCKS, (G, N_M, N_{GM}, K'_G, K_{GM}), N)\\ \rightarrow\quad \Diamond\!\!-\mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)\\ \quad \vee \neg(\Diamond\!\!-\ (\quad \mathsf{member\_requestkey}(M, GCKS, (G, N_M, K_{GM}), N)\\ \qquad \wedge\ \Diamond\!\!-\mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), \_)))\end{array}$$

Note that the definition of recency freshness is one of the few places we make use of round numbers, since the member requests and accepts the key in the same round. Note also that the GCKS's act of sending a key K_G protected by K'_G using the push protocol results in the expiration of K'_G.

4.4.2 Sequential Freshness for Pull Protocol

$$\begin{array}{l}\mathsf{member\_acceptpullkey}(M, GCKS, (G, N_M, N_{GM}, K_G, K_{GM}), \_)\\ \rightarrow\quad \Diamond\!\!-\mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)\\ \quad \vee \neg(\Diamond\!\!-\ (\quad \mathsf{member\_acceptkey}(M, GCKS, (G, K'_G), \_)\\ \qquad \wedge\ \Diamond\!\!-\ (\quad \mathsf{gcks\_createkey}(GCKS, (), (G, K'_G), \_)\\ \qquad\quad \wedge\ \Diamond\!\!-\mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_)))\end{array}$$

Recall from Section *4.2.3* that a group key is sent (and therefore made current) immediately after it is created by the GCKS.

4.4.3 Sequential Freshness for Push Protocol

$$\begin{array}{l}\mathsf{member\_acceptpushkey}(M, GCKS, (G, K_G, K'_G), \_)\\ \rightarrow \neg(\Diamond\!\!-\ (\quad \mathsf{member\_acceptkey}(M, GCKS, (G, K''_G), \_)\\ \qquad \wedge\ \Diamond\!\!-\ (\quad \mathsf{gcks\_createkey}(GCKS, (), (G, K''_G), \_)\\ \qquad\quad \wedge\ \Diamond\!\!-\mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_)))\end{array}$$

Note that we do not specify recency freshness as a requirement for the push protocol. This can be achieved, if desired, by including timestamps in the Security Association, but this is not a requirement of GDOI.

4.4.4 Freshness of a Member's Key Request

We now consider a freshness requirement from the GCKS's point of view. When the GCKS responds to a member's request with a key, it must be sure that this is a new request, not a replay of some old request. Since a member's request contains a nonce which is intended to be unique, we make this into a requirement that a GCKS should not have previously distributed a key to that member using that nonce.

Note that this freshness requirement can only be guaranteed for an honest member, since there is nothing preventing a dishonest member from replaying an old request and then participating in the protocol to obtain a key. Since honest members are the only ones we are interested in protecting anyway, this is not a problem for us. However, we need a way of distinguishing between honest and dishonest members. We do this by borrowing a trick from the NRL Protocol Analyzer specification language, and referring to principals as $\mathsf{member}(M, H)$ where H is a variable that can be instantiated to honest or dishonest. At this point we are only interest in $\mathsf{member}(M, \mathsf{honest})$:

$$\begin{array}{l}\mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K_G, N_M, N_{GM}, G, K_{GM}), \_)\\ \rightarrow\ \Diamond\!\!-\mathsf{gcks\_losepairwisekey}(GCKS, (), (M_{\mathsf{h}}, K_{GM}), \_)\\ \quad \vee\neg\Diamond\!\!-\mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K'_G, N_M, N'_{GM}, G, K_{GM}), \_)\end{array}$$

where $M_{\mathsf{h}} = \mathsf{member}(M, \mathsf{honest})$.

4.4.5 Freshness of Proof of Possession

Freshness requirements for Proof of Possession are more similar to two-party freshness requirements than some of the others we have visited. Since POPs are computed on nonces supplied by sender and receiver we require that, if a principal accepts a POP for two nonces, then it should not have accepted it previously. Since the POP is computed on the sender's and receiver's nonces, this can be enforced by requiring that the GCKS does not engage in a sendpullkey event based on the same nonces twice, and that a member does not engage in an member_acceptpullkey event based on the same nonces twice. Note that the GCKS's freshness requirement is similar, but somewhat stronger than, the requirement for freshness of a member's key request; it is not dependent on the pairwise key being uncompromised.

$$\begin{array}{l}\mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K_G, N_M, N_{GM}, G, K_{GM}), \_)\\ \rightarrow\neg\Diamond\!\!-\mathsf{gcks\_sendpullkey}(GCKS, M_{\mathsf{h}}, (K'_G, N_M, N_{GM}, G, K'_{GM}), \_)\end{array}$$

$$\begin{array}{l}\mathsf{member\_acceptpullkey}(M, GCKS, (K_G, N_M, N_{GM}, G, K_{GM}), \_)\\ \rightarrow\neg\Diamond\!\!-\mathsf{member\_acceptpullkey}(M, GCKS, (K'_G, N_M, N_{GM}, G', K'_{GM}), \_)\end{array}$$

where again $M_{\mathsf{h}} = \mathsf{member}(M, \mathsf{honest})$.

4.5 Secrecy Requirements

GDOI has one basic secrecy requirement, that keys should only be learned by members of the group. However, we may want to put other conditions on this requirement. For example, we may require that new members should not have access to old keys (*backward access control*), and that expelled members will not have access to any keys generated after they were expelled (*forward access control*). GDOI also allows for an option that provides a degree of protection against compromise of pairwise keys; it allows for the optional use of Diffie-Hellman to assure *perfect forward secrecy*: if a pairwise key is stolen, then the intruder should only be able to learn key encryption keys distributed after the event.

As we can see, the different secrecy requirements are not quite orthogonal, and they can interact with each other in different ways. For example, one would not want to waste time with perfect forward secrecy if one did not also have backwards access control. In general, it is assumed that it is more likely that a dishonest member will join the group than that a pairwise key shared between only two principals will be stolen. So it makes little sense to use perfect forward secrecy to protect old keys, if they could be compromised

by having a group key distributed to a dishonest principal. Likewise, requirements such as forward and backward access control should not only govern the effects of the distribution of keys, but other events such as the stealing of keys. For example, if members should no longer have access to new keys after leaving the group, then an intruder's stealing a key should not give it access to subsequent keys either.

Our solution to this problem is to define a number of conditions describing sequences of events that define the situation under which an intruder might learn a key. These conditions can then be mixed and matched to put together the appropriate requirement. We can then use the NPATRL logic to reduce the requirements to normal form, when necessary.

In the remainder of this section, we describe the various sequences. These include five "base cases" that describe some simple sequences of events that could lead to key compromise, as well as two recursively defined cases that describe forward access control without backward access control, and vice versa. We also give several examples showing how the various cases can be combined to produce different types of requirements.

4.5.1 The Base Cases

The five base cases are as follows:

BC1(K_G, G):

$$\mathsf{gcks\_losegroupkey}(GCKS, (), (G, K_G), \_)$$

This describes the a group key-encryption key being stolen.

BC2a(K_G, G):

$$\begin{aligned} &\Diamond(\ \ \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), N) \\ &\quad \wedge \Diamond \mathsf{gcks\_sendpullkey}(GCKS, M_\mathsf{d}, (G, \_, N_{GM}, K''_G, K_{GM}), \_)) \\ \wedge\, &\neg\Diamond(\ \ \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), N) \\ &\quad \wedge \Diamond \mathsf{gcks\_cancel}(GCKS, M_\mathsf{d}, (G, N_{GM}), \_)) \end{aligned}$$

where $M_\mathsf{d} = \mathsf{member}(M, \mathsf{dishonest})$. This describes a group key being distributed while a dishonest member is in the group. Note that it is in two parts. The first says that the dishonest member has joined the group; the second says that the member has not left it yet. In order to take care of the possibility of multiple joinings and leavings, we give both join and leave the same index N_{GM}, which uniquely identifies M's joining the group.

BC2b(K_G, G):

$$\Diamond \mathsf{gcks\_sendpullkey}(GCKS, M_\mathsf{d}, (G, \_, N_{GM}, K_G, K_{GM}), \_)$$

for $M_\mathsf{d} = \mathsf{member}(M, \mathsf{dishonest})$. This describes a group key K_G being distributed to a dishonest member via a pull protocol, that is, the dishonest member is being admitted to the group.

BC3a(K_G, G):

$$\begin{aligned} &\Diamond \mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_) \\ \wedge\, &\Diamond \mathsf{gcks\_sendpullkey}(GCKS, M, (G, \_, \_, K_G, K_{GM}), \_) \end{aligned}$$

This describes the result of a pairwise key being lost and a key being sent using that pairwise key.

BC3b(K_G, G):

$$\begin{aligned} \Diamond\ (\ \ &\mathsf{gcks\_sendpullkey}(GCKS, M, (G, \_, \_, K_G, K_{GM}), \_) \\ \wedge\, &\Diamond \mathsf{gcks\_losepairwisekey}(GCKS, (), (M, K_{GM}), \_)) \end{aligned}$$

This describes a pairwise key being lost and a key being sent using that pairwise key *after* the pairwise key is lost.

4.5.2 The Recursive Cases

There are two recursive cases. The first describes an intruder learning an old key after a later key has become current. The second describes the intruder learning a key before another key expires. We call these two cases "backward inference" and "forward inference."

BI(K'_G, G)

$$\begin{aligned} &\Diamond \mathsf{learn}(P, (), (K_G, G), \_) \\ \wedge\, &\Diamond(\ \ \mathsf{gcks\_createkey}(GCKS, (), (G, K_G), \_) \\ &\quad \wedge \Diamond \mathsf{gcks\_createkey}(GCKS, (), (G, K'_G), \_)) \end{aligned}$$

Note that when a new key is sent, the old key expires. And, we assume any (non-initial) key is sent in a push-key message as soon as it is created. Thus BI for K'_G describes an intruder learning a key K_G that became current after a key K'_G was current.

FI(K_G, G)

$$\begin{aligned} &\Diamond \mathsf{learn}(P, (), (K''_G, G), \_) \\ \wedge\, &\Diamond(\ \ \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K_G, K'_G), \_) \\ &\quad \wedge \Diamond \mathsf{gcks\_sendpushkey}(GCKS, (), (G, K''_G, K'''_G), \_)) \end{aligned}$$

FI describes an intruder learning a key K''_G that expired before a later key K_G was generated.

Backward Inference will be used to specify forward access control without backward access control: If an intruder learns a key K_G, then BI(K_G, G) will be listed among the set of possible paths to that event, but not FI(K_G, G), that is, the intruder may have learned K_G as a result of learning a key K'_G that expired previously to K_G, but not a key K^*_G that was generated after K_G expired. Similarly, Forward Inference will be used to specify backward access control without forward access control: if an intruder learns a key K_G, then FI(K_G, G) will be listed among the set of of possible paths to that event, but not BI(K_G, G).

We note that there appear to be some major changes from the original, informal, definition of forward and backward access control. The original definition put the requirement on the knowledge of any group member, not on the intruder. Also, the original requirement discussed a member learning a key as a result of joining the group, while we simply consider the results of the intruder learning a key without specifying how it was learned.

Our rationale for changing the focus from member to intruder can be expressed in two steps. In the NRL Protocol Analyzer model, we assume that dishonest group members can do everything honest group members do and more, since honest members can only obey the rules of the protocol. Thus any conditions on a dishonest member's learning a key should also hold for an honest member. Secondly, we assume that all dishonest members share information with the intruder, so that any conditions on the intruder's learning a key would imply the same condition for a dishonest member learning that information.

4.5.3 Sample Requirements

In this section, we show how the various "cases" can be combined into requirements.

Weak Secrecy

The weakest form of secrecy requirement simply requires that the protocol should protect against key compromise given the most benign assumptions possible: that is, that neither pairwise or key encryption keys have been lost, and no dishonest members have even joined the group. This can be described in terms of three separate conditions:

$$\begin{array}{l} \mathsf{learn}(P,(),(K_G,G),\_) \\ \rightarrow\ \mathsf{BC1}(K'_G,G)\ \vee\ \mathsf{BC2b}(K'_G,G)\ \vee\ \mathsf{BC3a}(K'_G,G) \end{array}$$

In other words, the intruder should not learn a key K_G for G unless some group key has previously been lost, a dishonest member joined the group at some time, or a pairwise key that was used to distribute a group key was stolen, either before or after being used.

Strong Secrecy

We can also use the base cases to formulate the strongest type of secrecy possible. In strong secrecy, the intruder learns a key K_G only if K_G is lost, a dishonest member received K_G, either when it joined the group or while it was a member of the group, or if a pairwise key was stolen and used to distribute K_G. We may or may not wish to require perfect forward secrecy.

Here, for example, is strong secrecy with perfect forward secrecy:

$$\begin{array}{l} \mathsf{learn}(P,(),(K_G,G),\_) \\ \rightarrow\quad \mathsf{BC1}(K_G,G)\ \vee\ \mathsf{BC2a}(K_G,G)\ \vee\ \mathsf{BC2b}(K_G,G) \\ \quad \vee \mathsf{BC3b}(K_G,G) \end{array}$$

Forward Access Control

Forward access control (without backward access control) can be thought of as strong secrecy together with added condition of backward inference: An intruder can learn a key, not only if the key was lost, distributed to a dishonest member, or distributed using a lost pairwise key, but if the key became current before the intruder learned a later key, *e.g.*, because a dishonest member joined the group. We do not include perfect forward secrecy, since protecting against old keys being compromised as a result of a stolen pairwise key makes no sense if the keys could be learned as a result of a dishonest member joining the group at any point:

$$\begin{array}{l} \mathsf{learn}(P,(),(K_G,G),\_) \\ \rightarrow\quad \mathsf{BC1}(K_G,G)\ \vee\ \mathsf{BC2a}(K_G,G)\ \vee\ \mathsf{BC2b}(K_G,G) \\ \quad \vee \mathsf{BC3a}(K_G,G)\ \vee\ \mathsf{BI}(K_G,G) \end{array}$$

Backward Access Control

Backward access control (without forward access control) can be specified similarly to forward access control, except that we replace backward with forward inference. We can require perfect forward secrecy or not. Here, for example, is backward access control without forward access control but with perfect forward secrecy:

$$\begin{array}{l} \mathsf{learn}(P,(),(K_G,G),\_) \\ \rightarrow\quad \mathsf{BC1}(K_G,G)\ \vee\ \mathsf{BC2a}(K_G,G)\ \vee\ \mathsf{BC2b}(K_G,G) \\ \quad \vee \mathsf{BC3b}(K_G,G)\ \vee\ \mathsf{FI}(K_G,G) \end{array}$$

If we wanted to omit the perfect forward secrecy requirement we would substitute $\mathsf{BC3a}(K_G,G)$ for $\mathsf{BC3b}(K_G,G)$.

In appendix A we show how to put the requirements for Forward and Backward Access Control into normal form.

5. CONCLUSIONS

We have presented a set of formal security requirements for the group protocol GDOI. In developing these requirements we learned much, not only about GDOI itself, but about the nature of requirements for open-ended cryptographic protocols. This has motivated us to develop the NPATRL requirements language into a full-scale logic that can be used to reason about and simplify requirements as well as specify them.

As this paper is being written, we are currently using the NRL Protocol Analyzer to verify that GDOI satisfies these requirements. This in turn may lead to a revision or improvement of the requirements as we discover more by our analysis. We have also been able to use our formalization of the requirements to discover and suggest improvements to GDOI. These suggestions have been incorporated into later versions of the draft. Thus, we have already found these requirements to be useful.

6. REFERENCES

[1] M. Baugher, T. Hardjono, H. Harney, and B. Weis. Group domain of interpretation for ISAKMP. available at `http://search.ietf.org/internet-drafts/draft-irtf-smug-gdoi-01.txt`, January 2001.

[2] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *Proc. of INFOCOM'99, vol. 2*, pages 708–716, March 1999.

[3] Brian F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[4] Danny Dolev and Andrew C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, March 1983. Preliminary version in *Proc. 22nd Annual IEEE Symp. Foundations of Computer Science,* 1981, 350–357.

[5] Naganand Doraswamy and Dan Harkins. *IPSEC: The New Security Standard for the Internet, Intranets, and Virtual Private Networks*. Prentice Hall, 1999.

[6] Robert Goldblatt. *Logics of Time and Computation, 2^{nd} edition*, volume 7 of *CSLI Lecture Notes*. CSLI Publications, Stanford, 1992.

[7] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, IETF, November 1998. available at `ftp://ftp.isi.edu/in-notes/rfc2409.txt`.

[8] G.E. Hughes and M.J. Creswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[9] C. Meadows and P. Syverson. A formal specification of requirements for payment transactions in the SET protocol. In R. Hirschfeld, editor, *Financial Cryptography, FC'98*, pages 122–140. Springer-Verlag, LNCS 1465, 1998.

[10] Catherine Meadows. A model of computation for the NRL Protocol Analyzer. In *Proceedings of the 7th Computer Security Foundations Workshop*, pages 84–89. IEEE CS Press, June 1994.

[11] Catherine Meadows. The NRL Protocol Analyzer: An

overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[12] Catherine Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9(1–2):143–164, 2001.

[13] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11(8), August 2000.

[14] P. Syverson and C. Meadows. A logical language for specifying cryptographic protocol requirements. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE CS Press, May 1993.

[15] P. Syverson and C. Meadows. Formal requirements for key distribution protocols. In A. De Santis, editor, *Advances in Cryptology — EUROCRYPT '94*, pages 32–331. Springer-Verlag, LNCS 950, 1994.

[16] P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1 and 2):27–59, January 1996.

APPENDIX

A. REMOVING RECURSION

In this appendix we show how we use the NPATRL logic to remove recursion from the requirements for forward and backwards access control. Removing recursion is not only desirable from the point of view of the NRL Protocol Analyzer, but also because a recursively defined condition may cause an infinite regression in other model checkers and theorem provers.

We present only the proof for backward access control; that for forward access control is similar.

LEMMA 1.
The backward access control condition $\mathsf{BAC}(K_G, G) =$

$$\begin{array}{l}\mathsf{learn}(P,(),(K_G,G),\_)\\ \quad\rightarrow\quad \Diamond\mathsf{BC1}(K_G,G)\ \vee\ \Diamond\mathsf{BC2a}(K_G,G)\ \vee\ \Diamond\mathsf{BC2b}(K_G,G)\\ \qquad\vee\Diamond\mathsf{BC3a}(K_G,G)\ \vee\ \Diamond\mathsf{FI}(K_G,G)\end{array}$$

is equivalent to the conditions $\mathsf{NRFAC}(K_G, G) =$

$$\begin{array}{l}\mathsf{learn}(P,(),(K_G,G),\_)\\ \quad\rightarrow\quad \Diamond\mathsf{BC1}(K_G,G)\ \vee\ \Diamond\mathsf{BC2a}(K_G,G)\ \vee\ \Diamond\mathsf{BC2b}(K_G,G)\\ \qquad\vee\Diamond\mathsf{BC3a}(K_G,G)\ \vee\ \Diamond\mathsf{NRFI}(K_G,G),\end{array}$$

where $\mathsf{NRFI}(K_G, G) =$

$$\begin{array}{l}(\mathsf{BC1}(K''_G,G)\vee\mathsf{BC2a}(K''_G,G)\vee\mathsf{BC2b}(K''_G,G)\vee\Diamond\mathsf{BC3a}(K''_G,G))\\ \wedge\,\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_G,K'_G),\_)\\ \wedge\,\Diamond\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K''_G,K'''_G),\_).\end{array}$$

PROOF. We make use of the following facts that follow from the NPATRL axioms. For reasons of space, we leave the proofs as an exercise to the reader:

1. $\Diamond(A\ \wedge\ B)\rightarrow\Diamond A\ \wedge\ \Diamond B$;
2. $\Diamond\Diamond A\rightarrow\Diamond A$, and;
3. $(A\ \wedge\ \Diamond B)\ \wedge\ (C\ \wedge\ \Diamond A)\rightarrow C\ \wedge\ \Diamond B$.

For purposes of of this proof, let $\mathsf{BC}(K, G) =$

$$\Diamond\mathsf{BC1}(K,G)\ \vee\ \Diamond\mathsf{BC2a}(K,G)\ \vee\ \Diamond\mathsf{BC2b}(K,G)\ \vee\ \Diamond\mathsf{BC3a}(K,G).$$

We need to show that "$\mathsf{learn}(P,(),(K,G),\_)\rightarrow\mathsf{BC}(K,G)\ \vee\ \Diamond\mathsf{NRFI}(K,G)$" is logically equivalent to "$\mathsf{learn}(P,(),(K,G),\_)\rightarrow\mathsf{BC}(K,G)\ \vee\ \Diamond FI(K,G)$". It is clear that "$\Diamond\mathsf{BC}(K,G)\ \vee\ \Diamond\mathsf{NRFI}(K,G)$" implies "$\Diamond\mathsf{BC}(K,G)\ \vee\ \Diamond\mathsf{FI}(K,G)$", since "$(\mathsf{BC1}(K1,G)\ \vee\ \mathsf{BC2a}(K1,G)\ \vee\ \mathsf{BC2b}(K1,G)\ \vee\ \mathsf{BC3a}(K1,G)$" implies "$\mathsf{learn}(P,(),(K1,G),\_)$".

We prove the implication in the other direction by induction on the age of $K1$. Suppose that K is the first key used by the GCKS for G. Then, since there is no $K1$ that was distributed before K, neither "$\mathsf{FI}(K,G)$" nor "$\mathsf{NRFI}(K,G)$" holds, and so "$\mathsf{BC}(K,G)\ \vee\ \mathsf{FI}(K,G)$" is trivially equivalent to "$\mathsf{BC}(K,G)\ \vee\ NRFI(K,G)$".

Suppose that now that the result holds for the k'th key K_n used by the GCKS, for all $k < n$. Let K_n by the n'th key. Then

$$\begin{array}{l}\mathsf{learn}(P,(),Kn,G),\_)\\ \rightarrow\quad\Diamond\mathsf{BC}(K_n,G)\\ \qquad\vee(\ \ \mathsf{learn}(P,(),(K_k,G),\_)\\ \qquad\quad\wedge\,\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_n,K2),\_)\\ \qquad\quad\wedge\,\Diamond\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_k,K3),\_)),\end{array}$$

for some k. Since K_k was used before K_n, we have $k < n$, and by the induction hypothesis we get

$$\begin{array}{l}\mathsf{learn}(P,(),(K_n,G),\_)\\ \rightarrow\quad\mathsf{BC}(K_n,G)\\ \qquad\vee\Diamond(\ \ \Diamond(\ \ \mathsf{BC}(K_i,G)\\ \qquad\qquad\wedge\,\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_k,K'''),\_)\\ \qquad\qquad\wedge\,\Diamond\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_i,K''''),\_)))\\ \qquad\quad\wedge\,(\ \ \mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_n,K'),\_)\\ \qquad\qquad\wedge\,\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_k,K''),\_)))).\end{array}$$

Using the facts "$\Diamond(A\ \wedge\ B)\rightarrow\Diamond A\ \wedge\ \Diamond B$", that "$\Diamond\Diamond A\rightarrow\Diamond A$", and that "$(A\ \wedge\ \Diamond B)\ \wedge\ (C\ \wedge\ \Diamond A)\rightarrow C\ \wedge\ \Diamond B$", we have

$$\begin{array}{l}\mathsf{learn}(P,(),K_n,G),\_)\\ \rightarrow\quad\mathsf{BC}(K_n,G)\\ \qquad\vee\Diamond(\ \ \mathsf{BC}(K_i,G)\\ \qquad\qquad\wedge\,\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_n,K'''),\_)\\ \qquad\qquad\wedge\,\Diamond\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_i,K''''),\_)),\end{array}$$

which is the result we need. □

LEMMA 2.
The forward access control condition $\mathsf{FAC}(K_G, G) =$

$$\begin{array}{l}\mathsf{learn}(P,(),(K_G,G),\_)\\ \quad\rightarrow\quad \Diamond\mathsf{BC1}(K_G,G)\ \vee\ \Diamond\mathsf{BC2a}(K_G,G)\ \vee\ \Diamond\mathsf{BC2b}(K_G,G)\\ \qquad\vee\Diamond\mathsf{BC3a}(K_G,G)\ \vee\ \Diamond\mathsf{BI}(K_G,G)\end{array}$$

is equivalent to the conditions $\mathsf{NRBAC}(K_G, G) =$

$$\begin{array}{l}\mathsf{learn}(P,(),(K_G,G),\_)\\ \quad\rightarrow\quad \Diamond\mathsf{BC1}(K_G,G)\ \vee\ \Diamond\mathsf{BC2a}(K_G,G)\ \vee\ \Diamond\mathsf{BC2b}(K_G,G)\\ \qquad\vee\Diamond\mathsf{BC3a}(K_G,G)\ \vee\ \Diamond\mathsf{NRBI}(K_G,G),\end{array}$$

where $\mathsf{NRBI}(K''_G, G) =$

$$\begin{array}{l}(\mathsf{BC1}(K_G,G)\vee\mathsf{BC2a}(K_G,G)\vee\mathsf{BC2b}(K_G,G)\vee\Diamond\mathsf{BC3a}(K_G,G))\\ \wedge\,\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K_G,K'_G),\_)\\ \wedge\,\Diamond\mathsf{gcks\_sendpushkey}(GCKS,(),(G,K''_G,K'''_G),\_).\end{array}$$

PROOF. The proof is the same as for backward access control, except the base induction case is the most recent key instead of the first key, and the induction is on distance from the most recent key instead of on distance from the earliest key. □